Real Time Operating Systems

Real Time Operating Systems

Tutorial at SBCCI 2001

Prof. Dr. Franz J. Rammig

Literature

[Bu] **Giorgio C. Buttazzo:**
Hard Real-Time Computing Systems
- Predictable Scheduling Algorithms and Applications“
Kluwer Academic Publishers

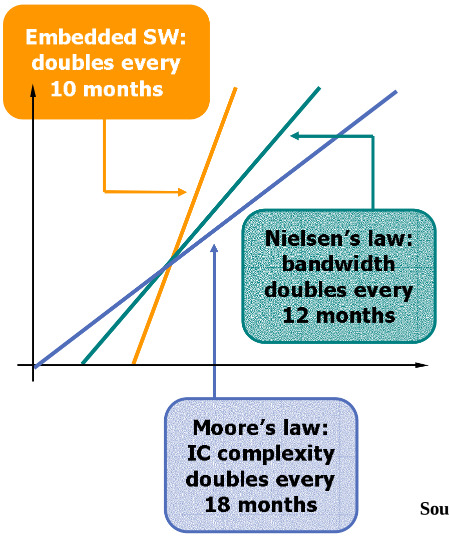
[Li] **Jane W. S. Liu:**
Real Time Systems
Prentice Hall

[Ko] **Hermann Kopetz:**
„Real-Time Systems: Design Principles for Distributed Applications“
Kluwer Academic Publishers

[Bu] **Alan Burns, Andy Wellings:**
„Real-Time Systems and Programming Languages“
Addison-Wesley

[La] **Phillip A. Laplante:**
„Real-Time Systems Design and Analysis - An Engineer‘s Handbook -“
IEEE Computer Society Press

The Challenge



Embedded SW: doubles every 10 months

Nielsen's law: bandwidth doubles every 12 months

Moore's law: IC complexity doubles every 18 months

Source: ST Microelectronics

Introduction

Kopetz:

A real-time Computer system is a computer system
in which the correctness of the system behaviour
depends not only
on the logical results of the computation,
but also
on the physical instant
at which these results are produced.

Where Can Real-Time Systems be Found?

RT-systems are everywhere:

- plant control
- control of production processes / industrial automation
- railway switching systems
- automotive applications
- flight control systems
- environmental acquisition and monitoring
- telecommunication systems
- robotics
- military systems
- space missions
- household appliances
- virtual / augmented reality

The Rising Software Intensity: Mobile Terminals

2nd Generation

Software

> 10 x

Intensity

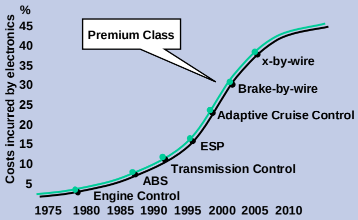
3rd Generation

2nd Generation		3rd Generation
4Mbit	**Memory**	**64Mbit**
30MIPS	**Radio Channel**	**200MIPS**
3-30MIPS	**Speech Codec**	**30MIPS**
-	**Voice Control**	**50MIPS**
-	**Video Codec**	**100MIPS**
8-16bits **10MHz**	**CTRL Processor** **Speed**	**16-32bits** **50MHz**

The Rising Software Intensity: Automobile

Electronic Content

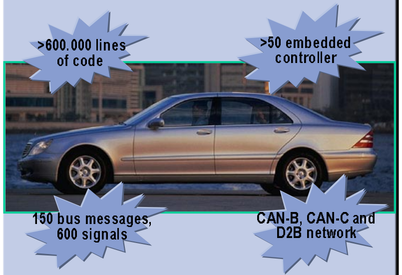
The continuous growth of vehicle electronics leads to a significant increase in software complexity



- **more than 80% of functions driven by software**
- **continuously increasing**

Premium Class, 2000

State of the Art



>600.000 lines of code

>50 embedded controller

150 bus messages, 600 signals

CAN-B, CAN-C and D2B network

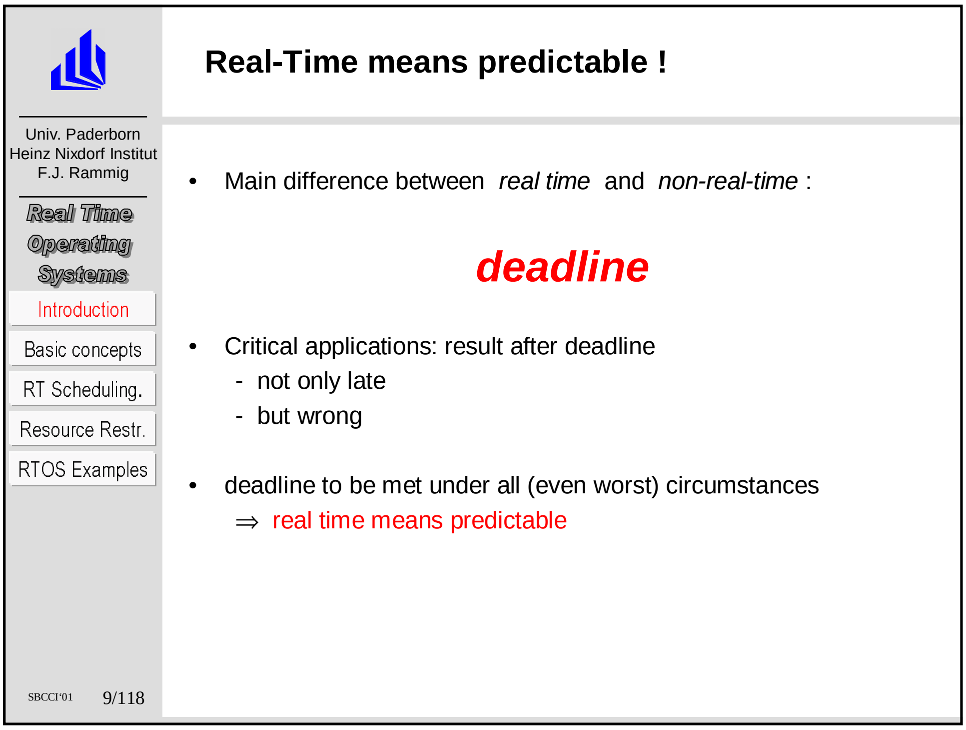
… not even accounting for telematics

Premium Class, 2000

What Does Real-Time Mean?

- Main difference to other computation: *time*
- *time* means that correctness of system depends
  - not only on logical results
  - but also on the time the results are produced
- *real* indicates: reaction to external events must occur during their evolution.

⇒ system time (*internal time*) has to be measured
with same time scale
as controlled environment (*external time*)



Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

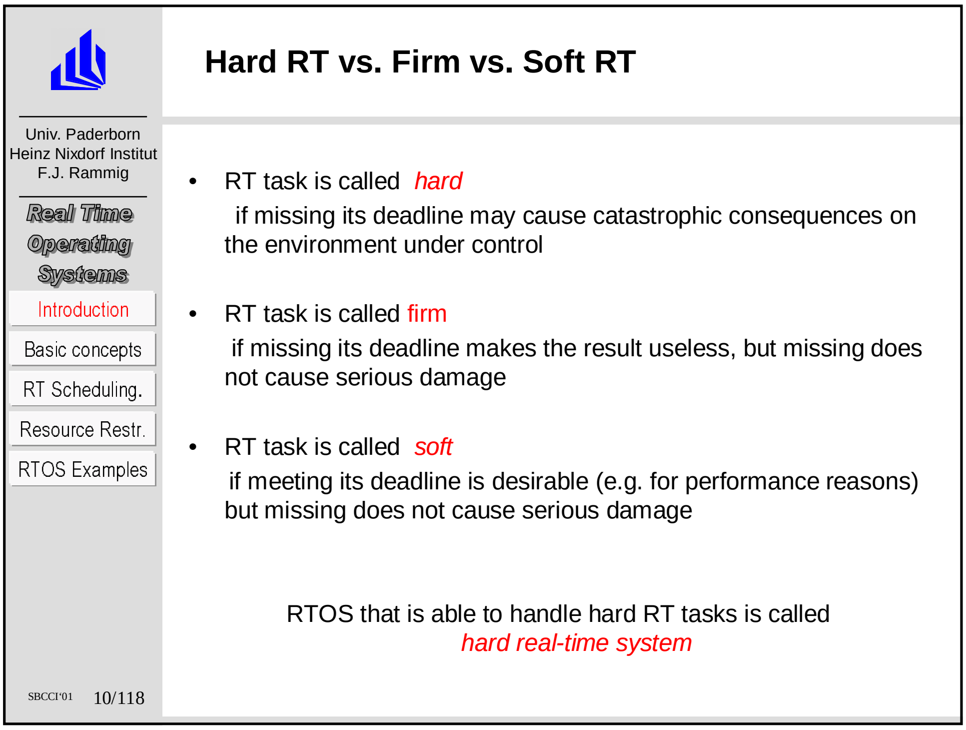
Real Time Operating Systems

Real-Time means predictable !

- Main difference between *real time* and *non-real-time* :

deadline

- Critical applications: result after deadline
  - not only late
  - but wrong
- deadline to be met under all (even worst) circumstances
  ⇒ real time means predictable



Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

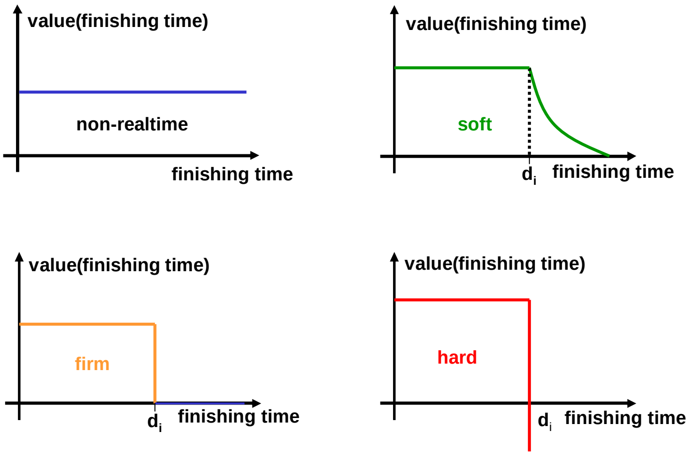
Real Time Operating Systems

Hard RT vs. Firm vs. Soft RT

- RT task is called *hard*
  if missing its deadline may cause catastrophic consequences on the environment under control
- RT task is called firm
  if missing its deadline makes the result useless, but missing does not cause serious damage
- RT task is called *soft*
  if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage

RTOS that is able to handle hard RT tasks is called
hard real-time system

Hard RT vs. Firm vs. Soft RT



Typical Hard RT Activities:

Typical Hard RT Activities:

- sensory data acquisition
- detection of critical conditions
- actuator servoing
- low-level control of critical system components

Typical application areas:

automotive : power-train control, air-bag control,
steer by wire, brake by wire
aircraft : engine control, aerodynamic control

Introduction
Basic concepts
RT Scheduling.
Resource Restr.
RTOS Examples

Typical Firm RT Activities:

Typical Firm RT Activities:

- decision support
- value prediction

Typical application areas:

Weather forecast
Decisions on stock exchange orders

Introduction
Basic concepts
RT Scheduling.
Resource Restr.
RTOS Examples

Typical Soft RT Activities:

Typical Soft RT Activities:

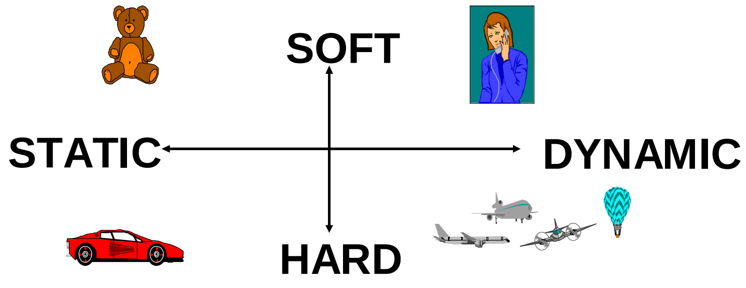
- command interpreter of user interface
- keyboard handling
- displaying messages on screen
- representing system state variables
- transmitting streaming data

Typical application areas:
- communication systems (voice over IP!)
- user interaction
- comfort electronics (body electronics in cars)

Real-Time ≠ Fast

~~Real-Time = Fast~~

RT = PREDICTABLE



SOFT

STATIC

DYNAMIC

HARD

Desirable Features of Real-Time Systems

- Timeliness
  - OS has to provide kernel mechanisms for
    - time management
    - handling tasks with explicit time constraints

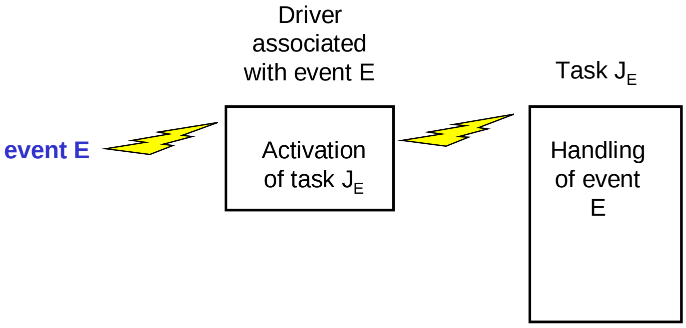
- Design for peak load
- Predictability
- Fault tolerance
- Maintainability

Achieving Predictability: Interrupt

Reduce the drivers to least possible size

- driver only activates proper task to take care of device
- this task executes under direct control of OS, just like any other task

=> control tasks than have higher priority than device task



Driver associated with event E

Task J_E

event E

Activation of task J_E

Handling of event E

(Source: [Bu])

Achieving Predictability: Semaphores

- Usual Semaphore mechanism not suited for real time applications:

  priority inversion problem

  (high priority task is blocked by low priority task)

- **Solution:**

  use special mechanism instead:
  - Priority Inheritance protocol
  - Priority Ceiling protocol

Basic Concepts

In this part we want to address the following aspects:

- Task constraints

- Scheduling problems

Introduction

- Basic SW entity handled by the OS:

Process

- Process = *task* in our context

- One processor, concurrent tasks ⇒

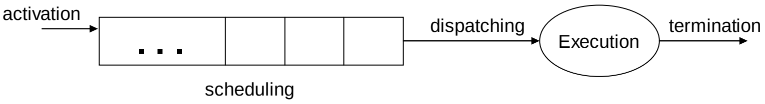
  - CPU has to be assigned to tasks
    - with respect to predefined criterion: *scheduling policy*
    - implementation of scheduling policy: *scheduling algorithm*
    - allocation of selected task to CPU: *dispatching*

Task Scheduling

- Three main states of tasks:

- *active* : can potentially execute
- *ready* : is waiting for CPU
- *running* : is executing on CPU

All ready tasks are kept in *ready queue*



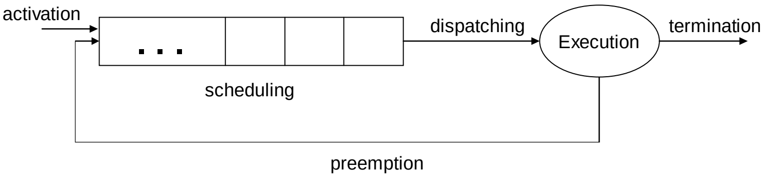
More than are ready queue possible!

(Source: [Bu])

Preemption

Running task may be interrupted at any point to allow
a more important task to gain the processor immediately.

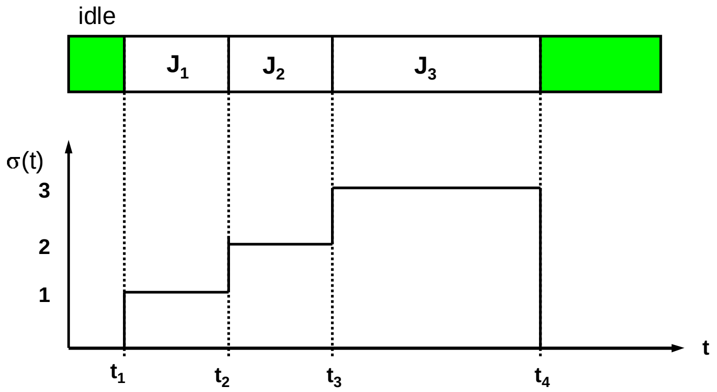
→ *preemption*



(Source: [Bu])

Schedule: Assigning Tasks to Processors

Assume three tasks J_1, J_2, J_3



Scheduling σ obtaind by executing three tasks J_1, J_2, and J_3

At times t_1, t_2, t_3, t_4 the processor performs a *context switch*

(Source: [Bu])

Feasible and Schedulable Sets of Tasks

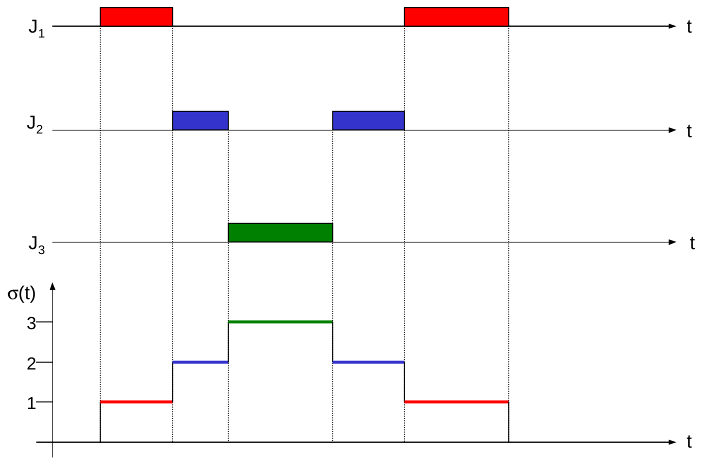
- Each interval $[t_i, t_{i+1})$ with σ (t) constant for $t \in [t_i, t_{i+1})$ is called *time slice*
- A *preemptive* schedule allows a running task to be suspended at any time. ⇒ tasks may be executed in disjoint intervals of time
- A schedule is called *feasible* if all tasks can be completed according to a set of specified constraints
- A set of tasks is called *schedulable* if there exist at least one algorithm that can produce a feasible schedule

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Example of Preemptive Schedule



(Source: [Bu])

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Types of Constraints

The following types of constraints are considered:

- Timing constraints
  meet your deadline
- Precedence constraints
  respect pre-requisites
- Resource constraints
  access only available resources

Timing Constraints

Real-time systems are characterized mostly by timing constraints

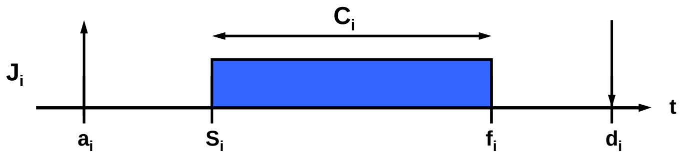
Typical timing constraint: *deadline*
deadline = time before which task has to be performed

Deadline missing separates two classes of RT systems:

- Hard : missing of deadline can cause catastrophic consequences
- Soft : missing of deadline decreases performance of system

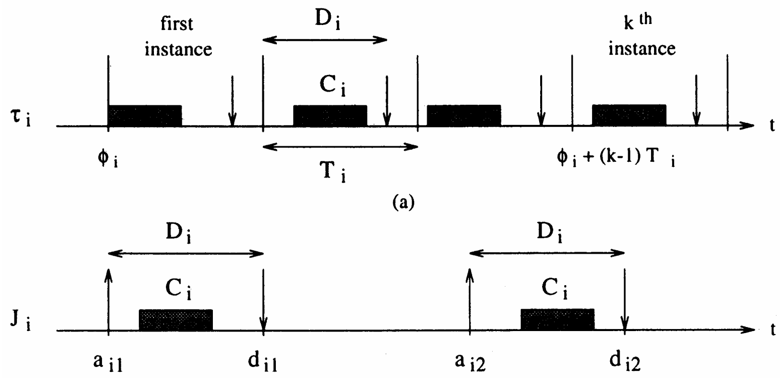
Parameters to Characterize RT-task J_i

- **Arrival time a_i :**
  the time J_i becomes ready for execution
  also called *request time* or *release time*, denoted by r_i
- **Computation time C_i:**
  time necessary for execution without interruption
- **Deadline d_i:**
  time before which task has to be completed its execution
- **Start time S_i:**
  time at which J_i start its execution
- **Finishing time f_i:**
  time at which J_i finishes its execution



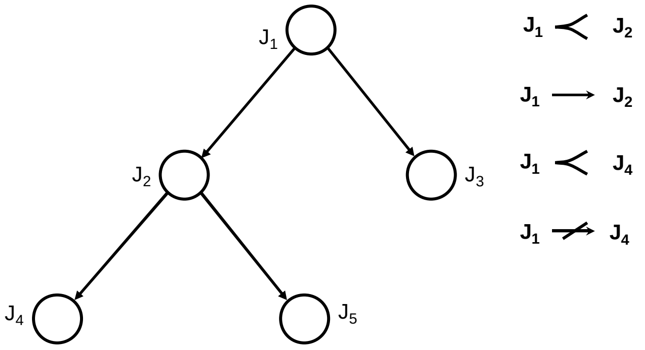
Typical parameters of a real-time task

Periodic and Aperiodic Tasks: Example



(Source: [Bu])

Precedence Relations (Example)



I_1 is called *beginning task*

I_4 and I_5 are called *ending tasks*

(Source: [Bu])

Resource Constraints

Process's view:
Resource is any SW structure to be used by process

Examples: data structure, set of variable, memory area file

private resource : dedicated to a particular process

shared resource : to be used by more than any process

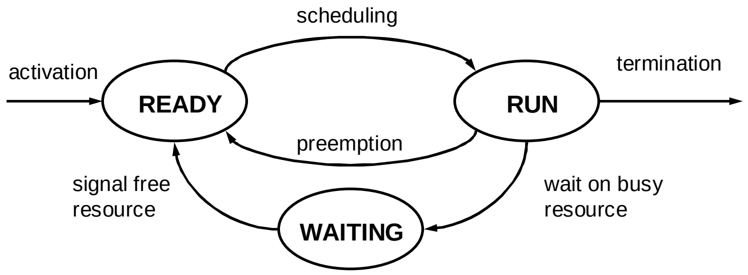
exclusive resource : shared resource where simultaneous access from different processes is not allowed

critical section : piece of code that is executed under mutual exclusion constraints. Management using semaphores (e.g.)

Blocked Tasks

A task waiting for an exclusive resource is called to be *blocked*

Processes blocked on the same resource are kept in a queue associated to the semaphore s protecting this resource. Signal(s) on s transfers the head of the queue to the ready state



Waiting state caused by resource constrains

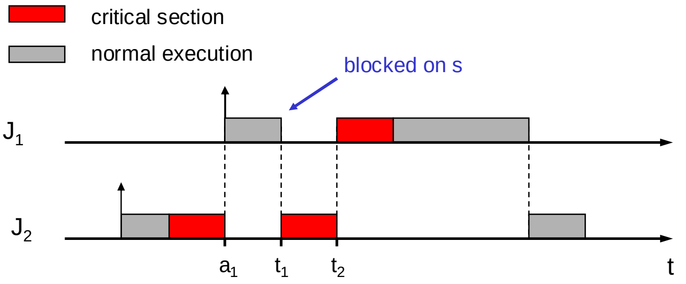
(Source: [Bu])

Priority Inversion

Assume preemption and two tasks J_1, J_2. Priority of J_1 may be higher than J_2.

If J_2 is started earlier it may enter its critical section.

It then may be preempted by J_1. However when J_1 wants to enter its critical section J_1 is blocked and J_2 is resumed.



Example of blocking on an exclusive resource

(Source: [Bu])

Scheduling

Given a set of n tasks $J = \{J_1, ..., J_n\}$
a set of m processors $P = \{P_1, ..., P_m\}$
a set of s resources $R = \{R_1, ..., R_s\}$

Precedences may be given using a precedence graph and timing constraints may be associated to each task.

Scheduling means to assign processors from P and resources from R to tasks from J in order to complete all tasks under the imposed constraints.

This problem is NP-complete !

Classification of Scheduling Algorithms (1)

- **Preemptive:** running task can be interrupted at any time.
- **Non-preemptive:** a task, once started is executed until completion.
- **Static:** scheduling decisions are based on fixed parameters (off-line) .
- **Dynamic:** scheduling decisions are based on parameters that change during system evolution.

Classification of Scheduling Algorithms (2)

- **Off-line :** Scheduling algorithm is performed on the entire task set before start of system . Calculated schedule is executed by *dispatcher*.
- **On-line :** scheduling decisions are taken at run-time every time a task enters or leaves the system.
- **Optimal :** the algorithm minimizes some given cost function, alternatively : it may fail to meet a deadline only if no other algorithm of the same class can meet it .
- **Heuristic :** algorithm that tends to find the optimal schedule but does not guarantee to find it.

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Guarantee-based Algorithms (1)

Hard real-time systems ⇒
highly predictable behavior,
i.e. feasibility of schedule has to be guaranteed in advance

⇒ System has to plan actions by looking ahead into the future, assuming a worst-case scenario

Static real-time systems (task set fixed & known à priori):

- all task activations can be pre-calculated *off-line*
- entire schedule can be stored in a table
- at runtime simple dispatching due to table takes place

\+ off-line very sophisticated algorithms possible

\- system is inflexible to environmental changes

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

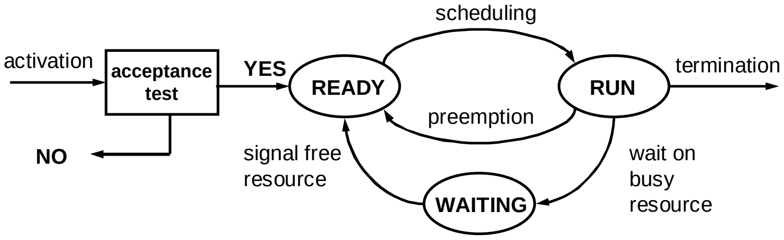
- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Guarantee-based Algorithms (2)

Dynamic real-time systems:

guarantee must be done *on-line* each time a new task enters the system

Typical scheme of guarantee mechanism for dynamic real-time systems:



Scheme of the guarantee mechanism used in dynamic hard real-time systems

(Source: [Bu])

Best-Effort Algorithms

Soft real-time systems:
only consequence of timing fault: degradation of system
example: video streaming (some jitter may be observed)
Adequate for soft real-time systems: *Best-Effort algorithms*

„tries its best“, e.g. tasks are queued due to time constraints, but no feasibility check is performed ⇒ tasks may be aborted

\+ best-efforts perform better in average (less overhead)
\- unpredictable, not suited for hard real-time applications

Real Time Scheduling

Aperiodic Scheduling
- Earliest Deadline First (EDF)
- Modified EDF

Periodic Scheduling
- Rate Monotonic Priority Assignment (RM)
- Earliest Deadline First (EDF)

Servers
- Fixed Priority
- Dynamic Priority

Aperiodic Scheduling: Earliest Deadline First (EDF)

Characterization:

- mono processor
- dynamic arrivals, preemption allowed
- minimize maximum lateness

Principle: Earliest Deadline First (EDF)

Horn‘s Algorithm

Theorem (Horn)

Given a set of n independent tasks with arbitrary arrival times, any algorithm that **at any instant** executes the task with the **earliest absolute deadline** among all the ready tasks **is optimal** with respect to minimizing the maximum lateness.

Complexity: per task: inserting a newly arriving task into an ordered list properly: O(n)

n task =>total complexity $O(n^2)$

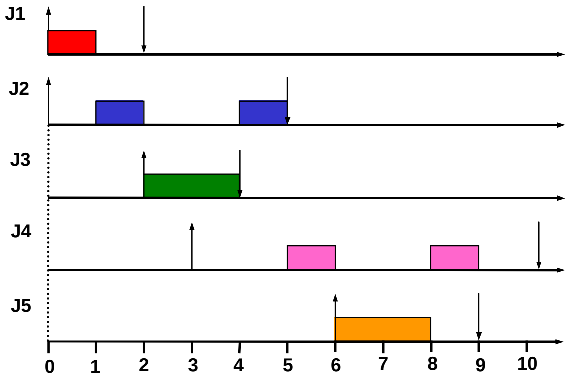
Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Example for EDF

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
c_i	1	2	2	2	2
d_i	2	5	4	10	9



(Source: [Bu])

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Scheduling with Precedence Constraints

- In General: NP-hard problem
- For special cases polynomial time algorithms possible

EDF with Precedence Constraints

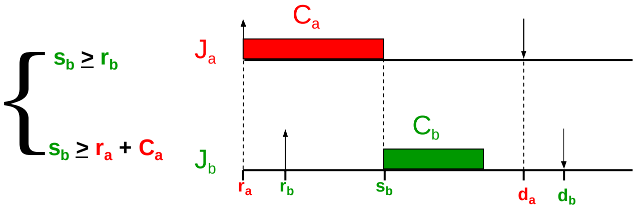
Assume *n* tasks with precedence constraints
and dynamic activations
Solvable in polynomial time only when pre-emption is allowed.

Solution of Chetto, Silly, and Bouchentouf:
Transform set J of dependent tasks into
set J* of independent ones by an adequate modification
of timing parameters
Then apply EDF.
The transformation ensures:
J* schedulable ⇔ J schedulable and all constraints satisfied
Modification of release times and deadlines such
that each task can not start
before its predecessors and cannot preempt their successors
(other tasks, however, may be preempted)

Modification of the Release Times

Given two tasks J_a and J_b , $J_a \rightarrow J_b$
⇒ the following two conditions must be satisfied:

- $s_b \geq r_b$ (J_b can not be started earlier than its release time)
- $s_b \geq r_a + c_a$ (J_b can not be started until J_a has finished)



If J_a --> J_b, then the release time of J_b can be replaced by max(r_b, $r_a + C_a$)

⇒ new release time for J_b: $r^*_b = \max(r_b, r_a + C_a)$ (Source: [Bu])

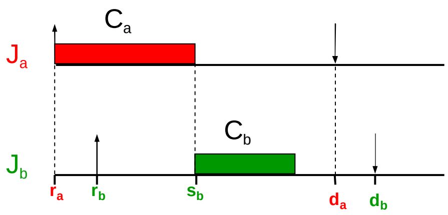
Modification of the Deadline

Given two tasks J_a and J_b , $J_a \rightarrow J_b$

⇒ in any feasible schedule that meets the precedences constraints the following two conditions must be satisfied:

- **$f_a \leq d_a$ (J_a must finish before its deadline)**
- **$f_a \leq d_b - c_b$ (J_a must finish before maximal start time of J_b)**

$$\begin{cases} s_b \geq r_b \\ s_b \geq r_a + C_a \end{cases}$$



If $J_a \rightarrow J_b$, then the deadline of J_a can be replaced by min(d_a, d_b + C_b)

new deadline for J_a: d_a^* = min (d_a , d_b - c_b)

(Source: [Bu])

Periodic Task Scheduling

Aspects to be addressed in this section:

- application areas of periodic tasks
- terms used in this context
- Rate Monotonic scheduling (RM)
- Earliest Deadline First scheduling (EDF)

Application Areas of Periodic Tasks

Periodic activities represent the major computational demand in many applications:

- sensory data acquisition
- low-level servoing
- control loops
- action planning
- system monitoring

Several periodic tasks running concurrently:

- individual timing constraints
  => each task
    - is regularly activated
    - at proper rate
    - is completed within its deadline

→ Multi Rate Systems

Basic Notations (1)

Γ set of periodic tasks

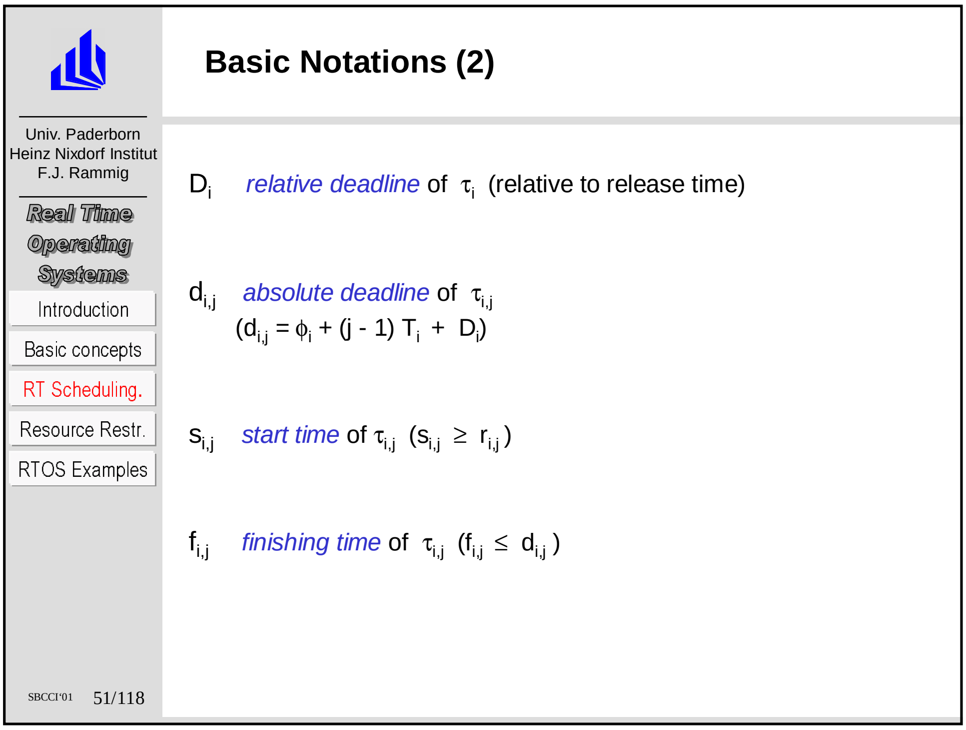
τ_i a generic periodic task

$\tau_{i\,j}$ instance j of task τ_i

$r_{i,j}$ *release time* of $\tau_{i,j}$

ϕ_i *phase of* τ_i (= $\tau_{i,1}$, i.e. release time of first instance)

T_i *period of* τ_i (= interval between two consecutive activations)



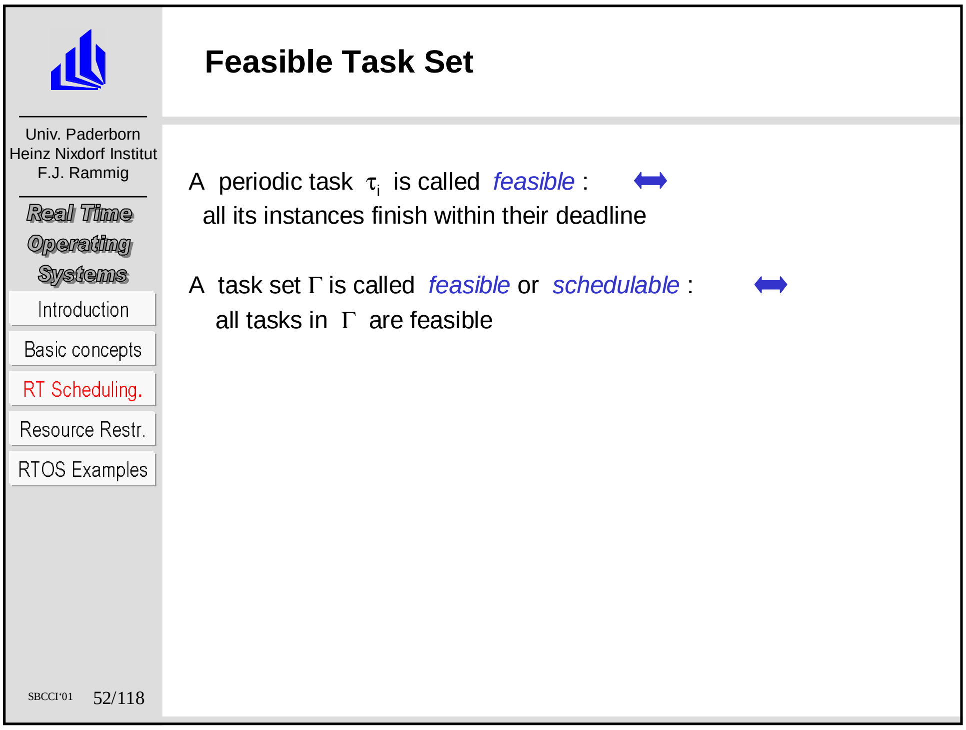
Basic Notations (2)

D_i *relative deadline* of τ_i (relative to release time)

$d_{i,j}$ *absolute deadline* of $\tau_{i,j}$
$(d_{i,j} = \phi_i + (j - 1) T_i + D_i)$

$s_{i,j}$ *start time* of $\tau_{i,j}$ $(s_{i,j} \geq r_{i,j})$

$f_{i,j}$ *finishing time* of $\tau_{i,j}$ $(f_{i,j} \leq d_{i,j})$



Feasible Task Set

A periodic task τ_i is called *feasible* : $\Leftrightarrow$
all its instances finish within their deadline

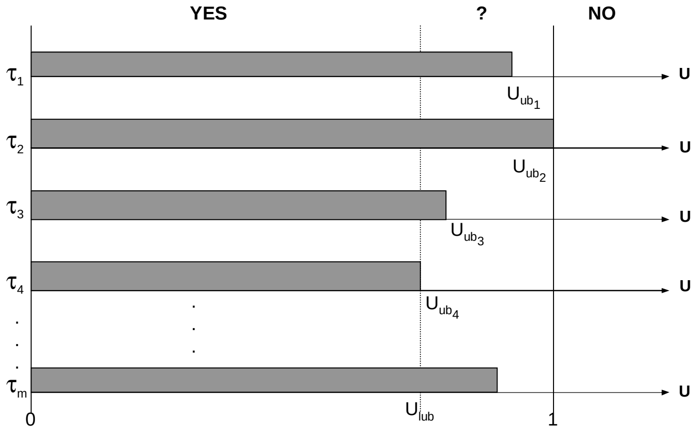
A task set Γ is called *feasible* or *schedulable* : $\Leftrightarrow$
all tasks in Γ are feasible

Processor Utilization Factor U

Given a set Γ of a periodic tasks the *processor utilization factor* U is the fraction of processor time spent in the execution of the taks set. C_i/T_i is the fraction of processor time spent in executing task τ_i

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Example for Least Upper Bound for U



(Source: [Bu])

Rate Monotonic Scheduling

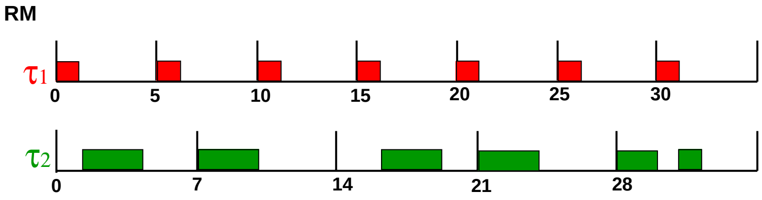
More precisely : **Rate Monotonic Priority Assignment** .
Priorities are assigned to tasks according to their request rates.

Tasks with higher request rates (i.e. shorter periods) get assigned higher priority.

Periods constant => RM is fixed-priority assignment
RM is intrinsically preemtive: currently executing task is preempted by a newly released task with shorter period.

Rate Monotonic Scheduling: Example

$\Gamma = \{\tau_1, \tau_2\}$, $T_1 = 5, C_1 = 1$, $T_2 = 7, C_2 = 3$, $U = 1/5 + 3/7 = 0.63$



Concluding remarks on RM

- RM is optimal among all fixed priority assignment
- RM guarantees that an arbitrary set of periodic tasks is schedulable if the total processor utilization U does not exceed the value of 0.69
- $U_{lub} = 0.69$ is sufficient but not necessary to guarantee the feasibility of a given task set.

Earliest Deadline First (EDF)

EDF algorithm:

- Dynamic scheduling rule
- Selects tasks according to their **absolute** deadlines
  - Tasks with earlier deadlines will be executed at higher priorities
- Absolute deadline of periodic task depends on current j-th instance:

$$d_{i,j} = \phi_i + (j - 1) T_i + D_i$$

- => EDF is dynamic priority assignment
- Intrinsically preemptive
- Applicable also for aperiodic tasks

EDF Schedulability Analysis

- Schedulability of periodic task set handled by EDF can be verified through the processor utilization factor
- U_{lub} here is 1 , i.e. 100 % utilization achievable

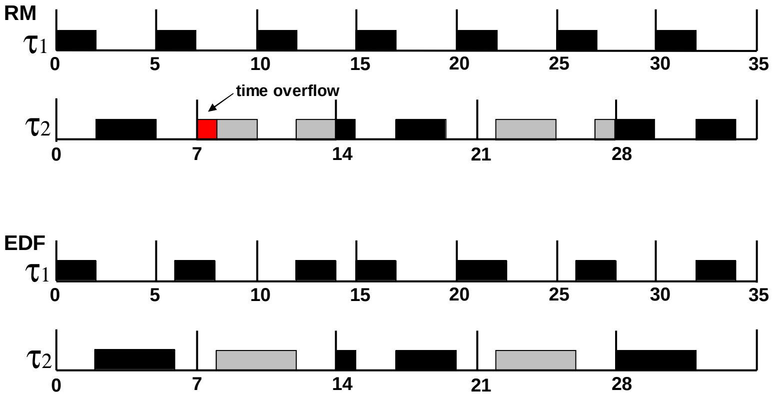
Theorem

A set of periodic tasks is schedulable with EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

EDF Schedulability Analysis : Example

U = 2/5 + 4/7 = 34/35 = 0.97 > ln 2 => not schedulable by RM



(Source: [Bu])

Periodic Task Scheduling: Summary

- Restriction on independent and preemptable periodic tasks
- Solution for fixed and dynamic priority assignments
- Rate Monotonic (RM) is optimal among fixed priority assignments
- Earliest Deadline First (EDF) is optimal among dynamic priority assignments
- Deadlines = Periods => guarantee test in O(n) using processor utilization, applicable to RM and EDF
- Deadlines < periodes => polynamical time algorithms for guarantee test
  - fixed priority: response time analysis
  - dynamic priority: processor utilization

Introduction to Servers

- Requirements in most real-time applications
  - periodic **and** aperiodic tasks
  - typically periodic tasks are time-driven, hard real-time
  - typically aperiodic tasks are event-driven, soft or hard RT

- Main objective for RT kernel:
  - guarantee hard RT tasks
  - provide good average response time for soft RT

- Aperiodic tasks are characterized by a minimum inter-arrival time, they are called *sporadic* ones

- Aperiodic tasks requiring on-line guarantee on individual instances are called *firm*

Fixed/Dynamic Priority Servers

- Handling of both, periodic and aperiodic tasks

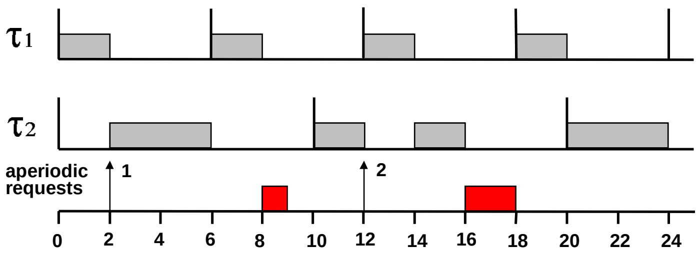
- Background Scheduling

- Periodic task as server for aperiodic tasks
  - Polling server
  - Deferrable server
  - Priority exchange
  - Sporadic server
  - Slack stealing

Background Scheduling

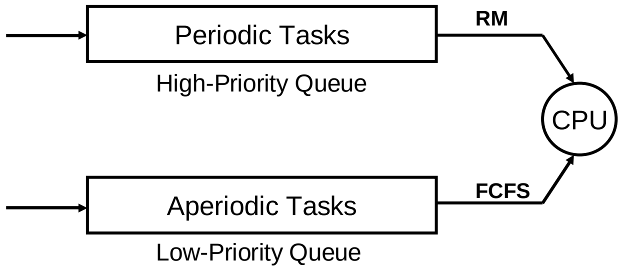
Simplest method:

Handle soft aperiodic tasks in the background behind periodic tasks, i.e. in the processor time left after scheduling all periodic tasks. Aperiodic tasks just get assigned a priority lower than any periodic one.



(Source: [Bu])

Organisation of Background Scheduling



Periodic Tasks

High-Priority Queue

Aperiodic Tasks

Low-Priority Queue

RM

FCFS

CPU

(Source: [Bu])

Example for Fixed Priority Server: Polling Server (1)

Idea to achieve more planable aperiodic task handling:

Let a specific periodic task service aperiodic requests.

This is called *server*, has (as any periodic task) a period T_s and a computation time C_s.

Computation time C_s is called *capacity* in this case.

The server is scheduled like any other periodic task, not necessarily at lowest priority.

Polling Server (2)

Polling Server (PS) is a special server with:

- By its period T_s, PS becomes active and serves any pending aperiodic requests within the limits of its capacity C_s
- No aperiodic request pending => PS suspends itself until beginning of its next period. Processor time is then used for periodic tasks
- If aperiodic task arrives just after suspension of PS it is served in the next period
- At the beginning of its period PS is replenished at its full value C_s

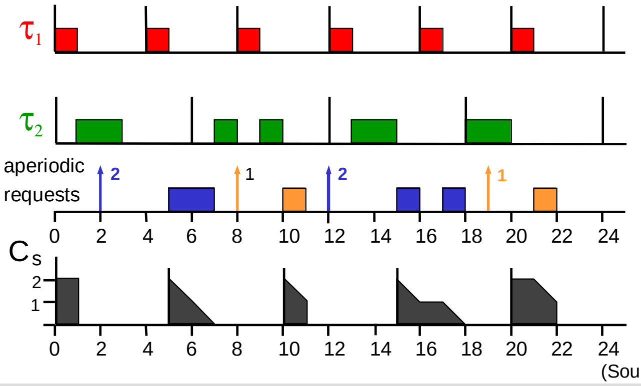
---

Example for Polling Server

	C_i	T_i
τ1	1	4
τ2	2	6

Server

$C_s = 2$
$T_s = 5$



(Source: [Bu])

Dynamic Priority Servers

Introduction

Recall: Dynamic scheduling algorithms have higher schedulability bounds than fixed priority ones

E.g.: Periodic task set with utilization factor $U_p = 0.6$.

If periodic tasks are served by EDF
the processor utilization bound goes up to 1.
=> maximum server size can reach $U_s = 1 - U_p = 0.4$

Assumptions

- All periodic tasks $\tau_i : i = 1, \dots, n$ have hard deadlines
- All aperiodic tasks $J_i : i = 1, \dots, m$ do not have hard deadlines
- Each periodic task τ_i has a period T_i, a computation time C_i, and a relative deadline D_i equal to its period ($D_i = T_i$)
- All periodic tasks are simultaneously activated at time $t = 0$
- Each aperiodic request has a known computation time but an unknown arrival time.

Dynamic Priority Exchange Server

Basic idea:
Let the server trade its runtime with the runtime of lower-priority periodic tasks (this means larger deadline under EDF) in case that there are no aperiodic requests pending.

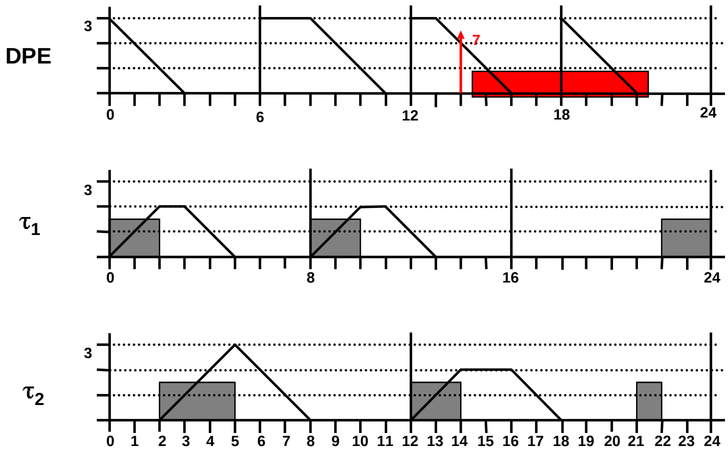
=> Server runtime may be exchanged with periodic tasks but not wasted (unless there are idle times). It is preserved at lower priority and can be reclaimed later when aperiodic requests enter the system.

Definition of DPE

Whenever the highest-priority entity in the system is an aperiodic capacity of C time units the following happens

- aperiodic requests in the system =>
  these are served until they complete or the capacity is exhausted (capacity used = execution time) using the server's capacity or a borrowed one
- no aperiodic request pending =>
  periodic task having the shortest deadline is executed and capacity equal to the length of the execution is added (i.e. borrowed) to the *aperiodic* capacity of the task deadline and subtracted from C > 0 at highest current priority (i.e. deadlines of the highest-priority capacity and the periodic tasks are exchanged)
- neither aperiodic requests nor periodic task instances are pending =>
  idle time, the capacity C is consumed until, at most, it is

DPE : Example



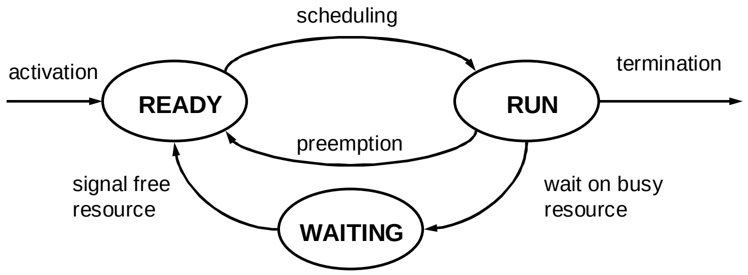
(Source: [Bu])

Resource Access Protocols

- *Resource:* any software structure that can be used by a process to advance its execution.

- Resource is said to be
  - *private* if it is dedicated to a particular process
  - *shared* if it is used by more than one task
  - *exclusive* it is shared but protected against concurrent accesses

Introduction and Terms



scheduling

activation

READY

RUN

termination

preemption

signal free resource

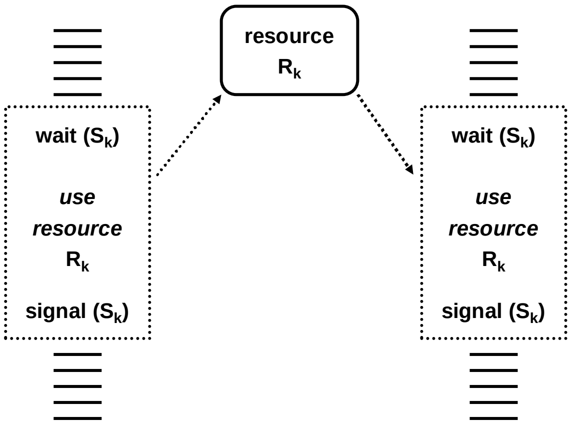
WAITING

wait on busy resource

(Source: [Bu])

The Priority Inversion Phenomenon (1)

Consider two tasks, sharing an exclusive resource



resource R_k

wait (S_k)

use resource
R_k

signal (S_k)

wait (S_k)

use resource
R_k

signal (S_k)

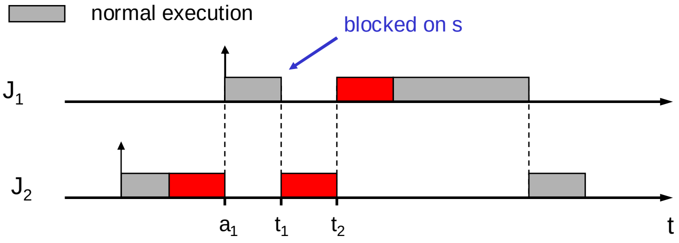
(Source: [Bu])

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

The Priority Inversion Phenomenon (2)

Assume that J_1 has higher priority than J_2.
Despite this fact J_1 may be blocked by J_2:

🟥 critical section
▨ normal execution



Example of blocking on an exclusive resource

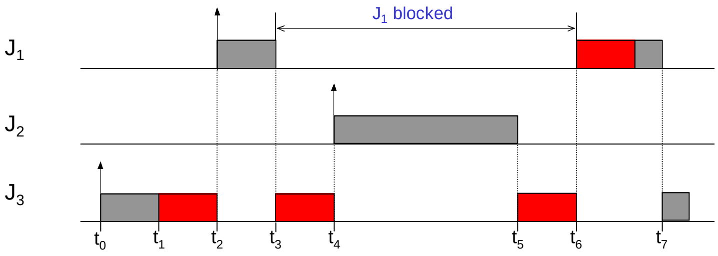
(Source: [Bu])

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

The Priority Inversion Problem (3)

Blocking time in the above example:
maximum blocking time of J_1 = duration of J_2 in critical section
→ unavoidable due to semantics of critical section
However: blocking time may be unbounded if there are tasks with intermediate priority:

▨ normal execution
🟥 critical section



Priority inversion occurs in interval $[t_3, t_6]$

(Source: [Bu])

Priority Inheritance Protocol

- Proposed by Sha, Rajkumar, Lehoczsky, 1990
- Basic idea:
  - when task J_i blocks one or more higher priority tasks, it temporarily inherits the highest priority of the blocked task
  - reason: this prevents medium-priority tasks from preempting J_i and by this prolonging the blocking period

---

PIP Definition (1)

- Jobs are scheduled based on their active priorities.
  Same priority => FCFS
- J_i tries to enter critical section $z_{i,j}$:
  case a) $R_{i,j}$ already held by lower-priority job:
  J_i is blocked (by the task that holds $R_{i,j}$)
  case b) otherwise: J_i enters $z_{i,j}$
- J_i blocked => it transfers its active priority to J_k ,
  the job that holds the semaphore
  => J_k resumes and executes the rest of its
  critical section with $p_k = p_i$
  i.e. J_k inherits the priority of J_i . In general,
  a task inherits the highest priority of the
  jobs blocked by it.

PIP Definition (2)

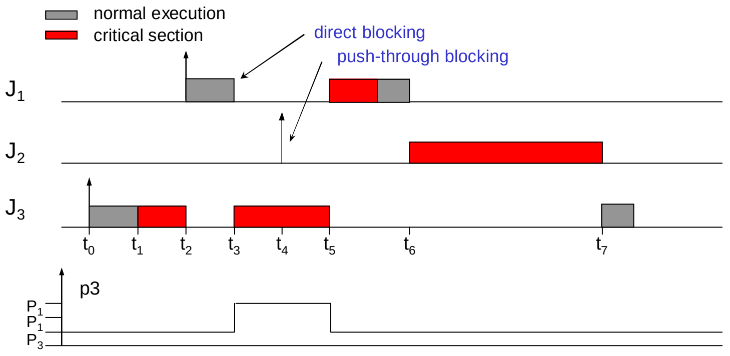
- J_k exits a critical section:
  - it unlocks the semaphore

  => highest priority job, blocked on this semaphore (if any) is awakened
  - the active priority of J_k is updated:
    - other jobs still blocked by J_k : J_k inherits the highest priority of the jobs blocked by J_k
    - otherwise J_k gets its normal priority
- Priority inheritance is transitive:

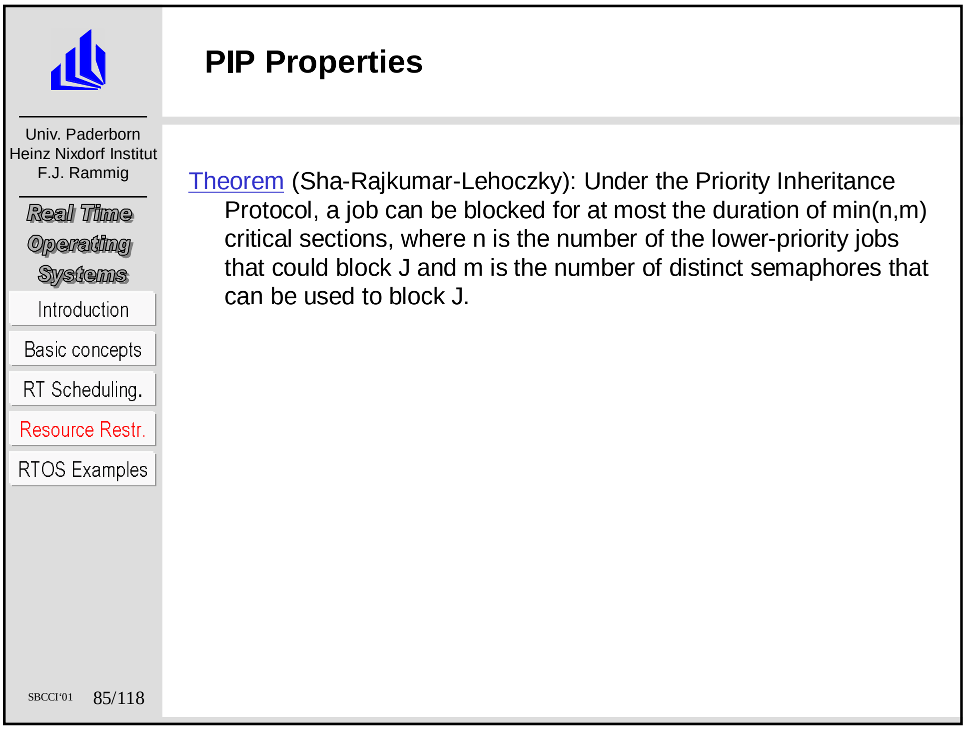
  if J_3 blocks J_2 and J_2 blocks J_1 then J_3 inherits the priority of J_1 via J_2

PIP: Example

Same situation as on slide 80, but now PIP applied



(Source: [Bu])



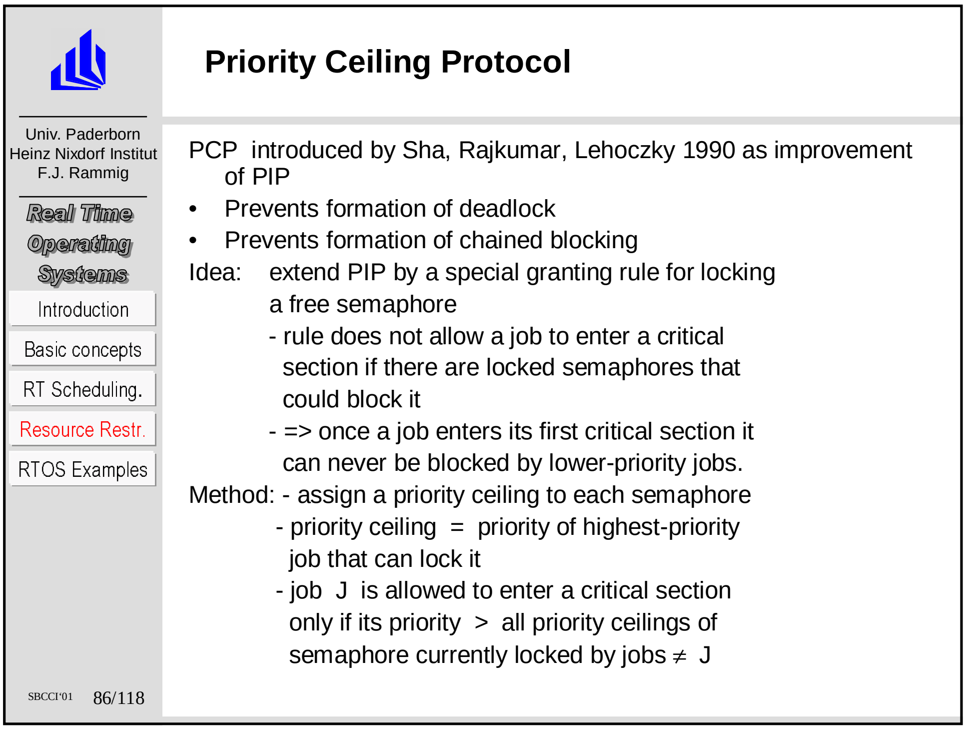
Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

PIP Properties

Theorem (Sha-Rajkumar-Lehoczky): Under the Priority Inheritance Protocol, a job can be blocked for at most the duration of min(n,m) critical sections, where n is the number of the lower-priority jobs that could block J and m is the number of distinct semaphores that can be used to block J.



Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

Priority Ceiling Protocol

PCP introduced by Sha, Rajkumar, Lehoczky 1990 as improvement of PIP

- Prevents formation of deadlock
- Prevents formation of chained blocking

Idea: extend PIP by a special granting rule for locking a free semaphore
- rule does not allow a job to enter a critical section if there are locked semaphores that could block it
- => once a job enters its first critical section it can never be blocked by lower-priority jobs.

Method: - assign a priority ceiling to each semaphore
- priority ceiling = priority of highest-priority job that can lock it
- job J is allowed to enter a critical section only if its priority > all priority ceilings of semaphore currently locked by jobs ≠ J

PCP Protocol Definition (1)

- Each semaphore S_k is assigned a priority ceiling $C(S_k)$ equal to the priority of the highest-priority job that can lock it. Note that $C(S_k)$ is a static value that can be computed off-line.
- Let J_i be the job with the highest priority among all jobs ready to run; thus, J_i is assigned the processor.
- Let S* be the semaphore with the highest ceiling among all the semaphores currently locked by jobs other than J_i and let C(S*) be its ceiling.
- To enter a critical section guarded by a semaphore S_k, J_i must have a priority higher than C(S*) . If $P_i \leq C(S^*)$, the lock on S_k is denied and J_i is said to be blocked on semaphore S* by the job that holds the lock on S*

PCP Protocol Definition (2)

- When a job J_i is blocked on a semaphore, it transmits its priority to the job, say J_k, that holds that semaphore. Hence, J_k resumes and executes the rest of its critical section with the priority of J_i. J_k is said to *inherit* the priority of J_i. In general, a task inherits the highest priority of the jobs blocked by it.
- When J_k exits a critical section, it unlocks the semaphore and the highest-priority job, if any, blocked on the semaphore is awakened. Moreover, the active priority of J_k is updated as follows: if no other jobs are blocked by J_k , p_k is set to the nominal priority P_k ; otherwise, it is set to the highest priority of the jobs blocked by J_k .
- Priority inheritance is transitive; that is, if a job J_3 blocks a job J_2, and J_2 blocks J_1 , then J_3 inherits the priority of J_1 via J_2 .

PCP Example (1)

Consider: 3 jobs J_0, J_1, J_2 with decreasing priorities

J_0 sequentially accesses critical sections guarded by S_0, S_1

J_1 accesses only c.s. guarded by S_2

J_2 uses S_2 and than a nested-in access to S_1

This results in the following priority ceiling of the semaphores:

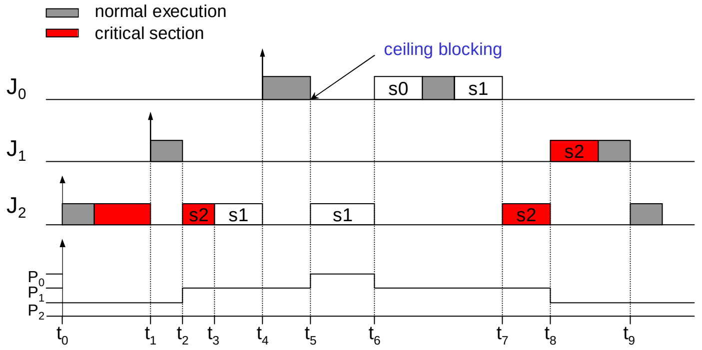
$C(S_0) = P_0$

$C(S_1) = P_0$

$C(S_2) = P_1$

(Source: [Bu])

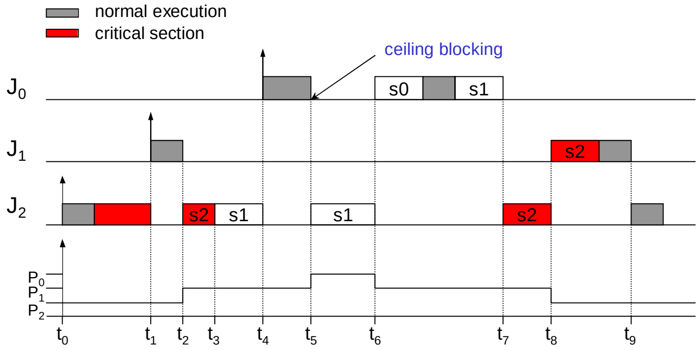
PCP Example (2)



- At time t_0, J_2 is activated and, since it is the only job ready to run, it starts executing and later locks semaphore S_2 .
- At time t_1 , J_1 becomes ready and preempts J_2 .
- At time t_2 , J_1 attempts to lock S_2, but it is blocked by protocol because P_1 it is not greater than $C(S_2)$. Then, J_2 inherits the priority of J_1 and resumes is execution .
- At time t_3 , J_2 successfully enters its nested critical section by locking S_1 . Note that J_2 is allowed to lock S_1 because no semaphore are locked by other jobs.

(Source: [Bu])

PCP Example (3)



- At time t_4 ,while J_2 is executing at a priority $p_2 = P_1$, J_0 becomes ready and preempts J_2 because $P_0 > p_2$.
- At time t_5 , J_0 attempts to lock S_0 , which is not locked by any job. However, J_0 is blocked by the protocol because its priority is not higher than $C(S_1)$, which is the highest ceiling among all semaphores currently locked by the other jobs. Since S_1 is locked by J_2 , J_2 inherits the priority of J_0 and resumes its execution.

(Source: [Bu])

PCP Properties

Theorem: The priority Ceiling Protocol prevents deadlocks.

Theorem: (Sha-Rajkumar-Lehoczky) Under the Priority Ceiling Protocol, a job J_i can be blocked for at most duration of one critical section .

FB17
Praktische Informatik
Prof. F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

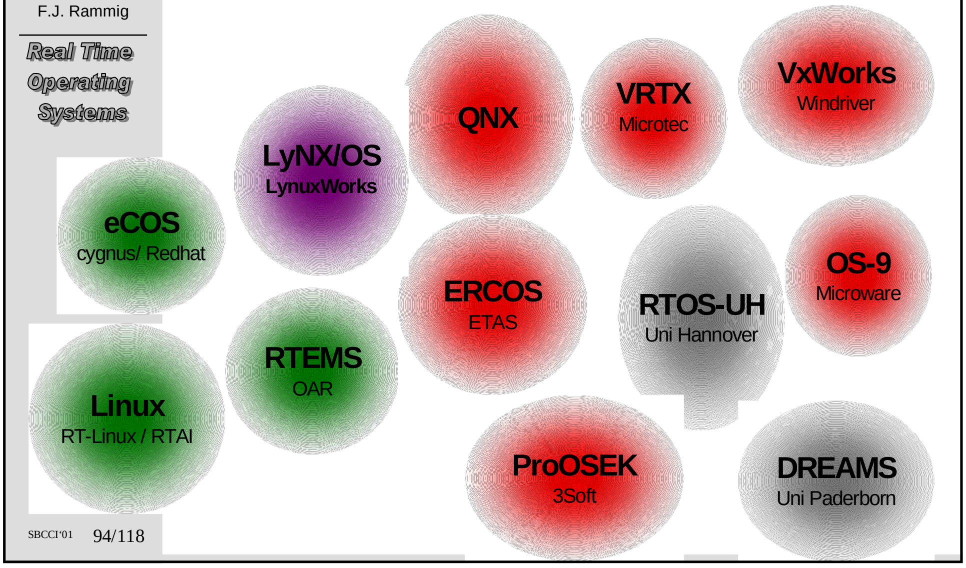
Examples of Real Time Operating Systems

- **Linux based**
- **Unix based**
- **QNX**
- **Others**

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

There is a Big Choice !



QNX

VRTX
Microtec

VxWorks
Windriver

LyNX/OS
LynuxWorks

eCOS
cygnus/ Redhat

ERCOS
ETAS

RTOS-UH
Uni Hannover

OS-9
Microware

RTEMS
OAR

Linux
RT-Linux / RTAI

ProOSEK
3Soft

DREAMS
Uni Paderborn

RTOS Based on Linux

RT-Linux and RTAI

hard real-time operating systems

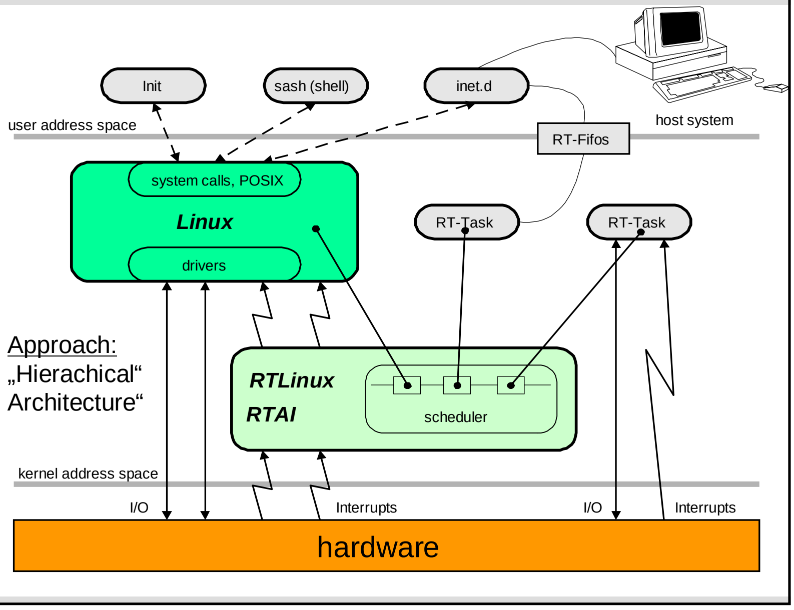
run Linux as its **lowest** priority execution thread

communication between RT threads and Linux with RT-FIFO

RT tasks running in kernel space

RTAI has more features
<=>
RT-Linux is more conservative

Make Linux Hard Real Time



Approach:
„Hierachical“
Architecture“

RT-Linux Characteristics

Development Hosts: Linux

Supported Target Processors: x86, PowerPC, MIPS, Alpha

Supported Compilers: gcc

Supported Networks: TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, X25, RPC, Telnet, Bootp

Supported Standards: ISO C, POSIX 1003.13, POSIX.1b, 1c, 1d, 1j subset

RTOS Supplied as: Source

Supported GUI: X-Windows, RTiC, LabView, Matlab

Available Components: Floating Point, Communication, Math Library, File Support

---

RT-Linux Technical Info

Typical Thread Switch Latency: depends on CPU

Guaranteed Maximum Interrupt Latency: <20us (CPU-dependent)

System Clock Resolution: <1 us

Priority Inversion Avoidance Mechanism: Yes, lock-free data structures, priority ceiling

Multiprocess Support: Yes

Multiprocessor Support: Yes

MMU Support: Yes

Scheduling Policies: Prioritized FIFO,extensible scheduler , EDF, Rate Monotonic

Royalty Free: Yes

RTAI Description

- RTAI is a hard realtime operating system
- RTAI considers Linux as a background task running when no real time activity occurs
- RTAI offers the same services of the Linux kernel core, adding the features of an industrial
real time operating system
- It is not an intrusive modification of the kernel; it uses the concept of HAL (*hardware abstraction layer*)

RTAI Characteristics

Development Hosts: Linux
Supported Target Processors: x86
Supported Compilers: gcc
Supported Networks: TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, X25, RPC, Telnet, Bootp
Supported Standards: POSIX 1003.1c
RTOS Supplied as: Source
Supported GUI: X-Windows
Available Components: Floating Point, Communication, File Support

RTOS Based on Linux

RED Linux and KURT

both are modifications of the standard kernel to add real-time characteristics

real-time tasks can directly access Linux kernel features

RED Linux

hard real-time operating system

additional kernel preemption points to reduce kernel latency

general real-time scheduling framework

KURT

soft real-time operating system

tasks are loadable modules

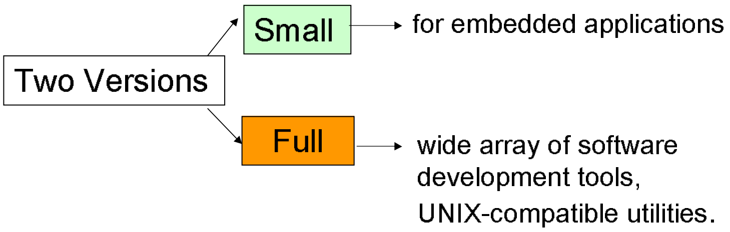
RTOS Based on UNIX

LynxOS

specifically designed for hard real-time applications

includes the 4.4 BSD system call interfaces and libraries and standard POSIX 1003.1, 1b, 1c

designed for complex real-time applications that require fast and deterministic response



Two Versions

Small → for embedded applications

Full → wide array of software development tools, UNIX-compatible utilities.

Introduction
Basic concepts
RT Scheduling.
Resource Restr.
RTOS Examples

LynxOS Description

- LynxOS is a UNIX-compatible, POSIX-conforming, multiprocess, and multithreaded RTOS
- Modular Design
- Includes the 4.4 BSD system call interfaces and libraries

Introduction
Basic concepts
RT Scheduling.
Resource Restr.
RTOS Examples

LynxOS Characteristics

Development Hosts: Sun Solaris, SunOS, RS6000, LynxOS Native/Hosted

Supported Target Processors: x86, 68k, PPC, microSPARC, microSPARC II, PA-RISC

Supported Compilers: Included in LynuxWorks Open Development Environment: gcc, G++; Via third-parties: FORTRAN 77/90, C++, Ada83, Ada95, Pascal, Modula-2

Supported Networks: TCP/IP, SNMP, NFS, Numerous network interface cards and devices, Other protocols and hardware through third-parties

Supported Standards: POSIX.1/.1b/.1c, Unix BSD 4.3

RTOS Supplied as: Object, Source

Supported GUI: X-Windows, Motif, others

Available Components: Floating Point, Communication, Math Library, File Support, Cache Support, Network Su

QNX

Two versions: QNX4 and QNX Neutrino
complete stand-alone RTOS
several features implemented like POSIX, IPC, …

QNX Neutrino is for embedded systems and QNX4 is for desktop

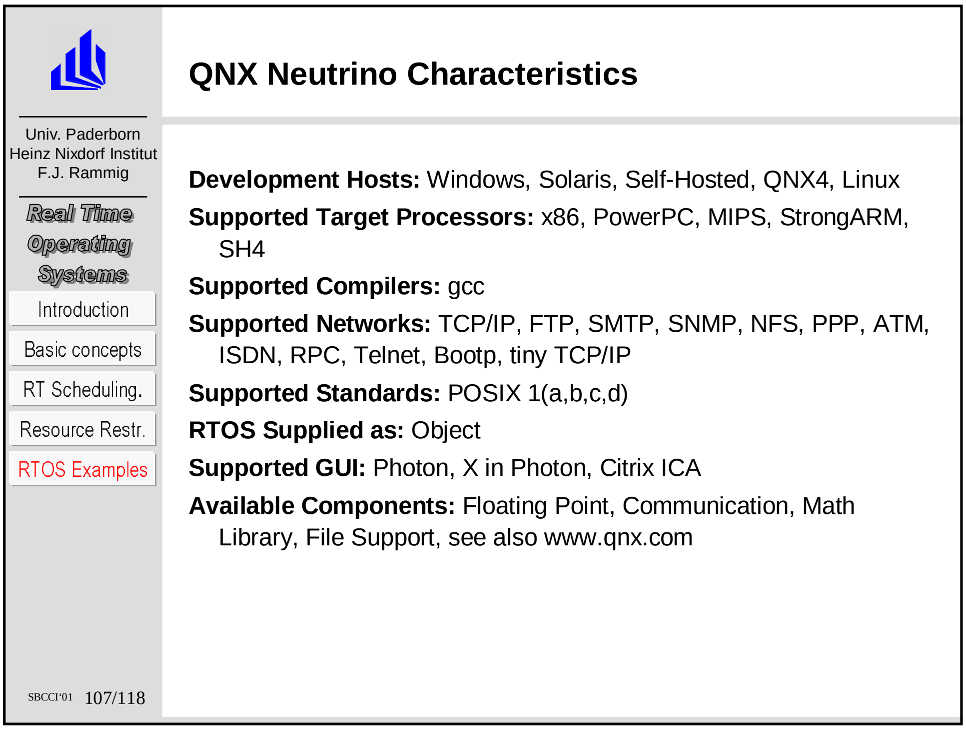
QNX4
- small, scalable, extensible, and fast microkernel
- can run in a distributed network of several hundred processors

QNX Neutrino
- possibility to create a single multi-threaded image for small embedded systems

---

QNX Neutrino Description

- QNX Neutrino microkernel delivers core realtime services for embedded applications
- QNX Neutrino is engineered for the latest POSIX 1003.1 standards and drafts including realtime and thread options
- QNX can be smoothly extended to support POSIX message queues, file systems, networking, and other OS-level capabilities with plug-in, service-providing modules.



QNX Neutrino Characteristics

Development Hosts: Windows, Solaris, Self-Hosted, QNX4, Linux

Supported Target Processors: x86, PowerPC, MIPS, StrongARM, SH4

Supported Compilers: gcc

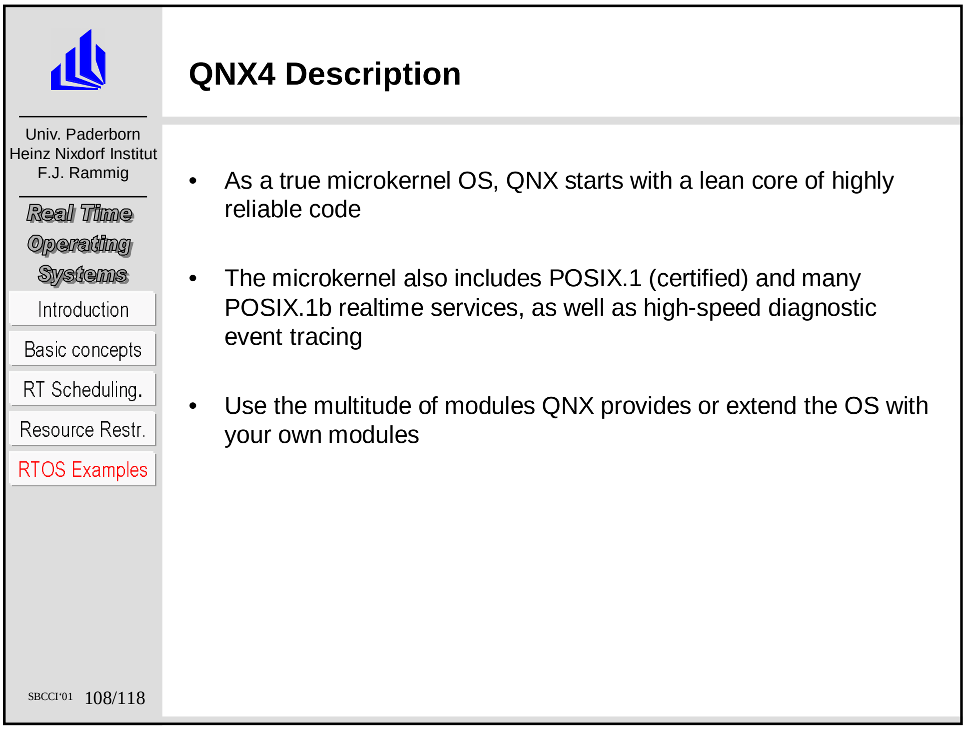
Supported Networks: TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, RPC, Telnet, Bootp, tiny TCP/IP

Supported Standards: POSIX 1(a,b,c,d)

RTOS Supplied as: Object

Supported GUI: Photon, X in Photon, Citrix ICA

Available Components: Floating Point, Communication, Math Library, File Support, see also www.qnx.com



QNX4 Description

- As a true microkernel OS, QNX starts with a lean core of highly reliable code
- The microkernel also includes POSIX.1 (certified) and many POSIX.1b realtime services, as well as high-speed diagnostic event tracing
- Use the multitude of modules QNX provides or extend the OS with your own modules

QNX 4 Characteristics

Development Hosts: Self-Hosted

Supported Target Processors: x86

Supported Compilers: Watcom

Supported Networks: TCP/IP, FTP, SMTP, SNMP, NFS, PPP, ATM, ISDN, RPC, Telnet, Bootp

Supported Standards: POSIX

RTOS Supplied as: Object

Supported GUI: Photon, X in Photon, Citrix ICA

Available Components: Floating Point, Communication, Math Library, File Support, see also www.qnx.com

---

Others

RTEMS
- high performance environment for embedded (military) applications
- kernel supports a notion of deadlines
- hard real-time

On Time RTOS32
- Windows NT subset
- fully integrates with Microsoft Visual Studio
- hard real-time

VxWorks
- Intended to embedded applications
- run-time component of the Tornado® II embedded development platform
- the user can create small and big configurations for the available hardware

VxWorks Description

- Run-Time component of the Tornado® II embedded development platform
- Intended to embedded aplications
- Comprises the core capabilities of the wind® microkernel along with advanced networking support, powerful file system and I/O management, and C++ and other standard run-time support
- Consists of development software and run-time software

VxWorks Characteristics

Development Hosts: DIGITAL UNIX

Supported Target Processors: x86, PowerPC, ARM, MIPS, 68K, CPU 32, ColdFire, MCORE, Pentium,i960,SH, SPARC, NEC V8xx, M32 R/D, RAD6000, ST 20, TriCore

Supported Compilers:

Supported Networks: TCP/IP, FTP, SMTP, NFS, PPP, RPC, Telnet, BSD 4.4 TCP/IP networking,IP, IGMP, CIDR, TCP, UDP, ARP, RIP v.1/v.2, Standard Berkeley sockets, zbufs (a.k.a., zero-copy sockets),SLIP, CSLIP, BOOTP, DNS, DHCP, TFTP, NFS, ONC RPC, WindNet SNMP v.1/v.2c with MIB compiler - optional, WindNet OSPF

Supported Standards: POSIX

RTOS Supplied as: Object

Supported GUI:

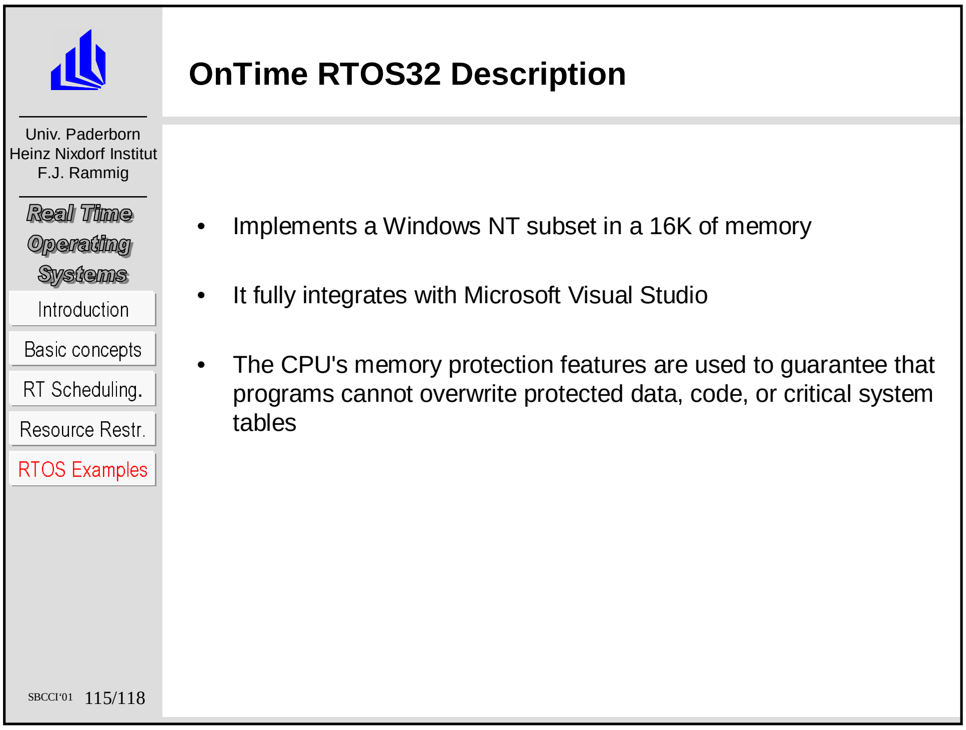
Available Components: IPC, remote login tool, File services, shell

RTEMS Description

- Was developed by U.S Army
- High performance environment for embedded military application including many features
- This kernel support a notion of deadlines
- Further OS modules can be added

RTEMS Characteristics

Development Hosts:
Supported Target Processors: MC68xxx, Hitachi SH, i386, MIPS, PowerPC, SPARC, AMD A29K, HP PA-RISC
Supported Compilers: gcc
Supported Networks: TCP/IP Stack
Supported Standards: POSIX 1003.1b
RTOS Supplied as: Object
Supported GUI:
Available Components: IPC, File System



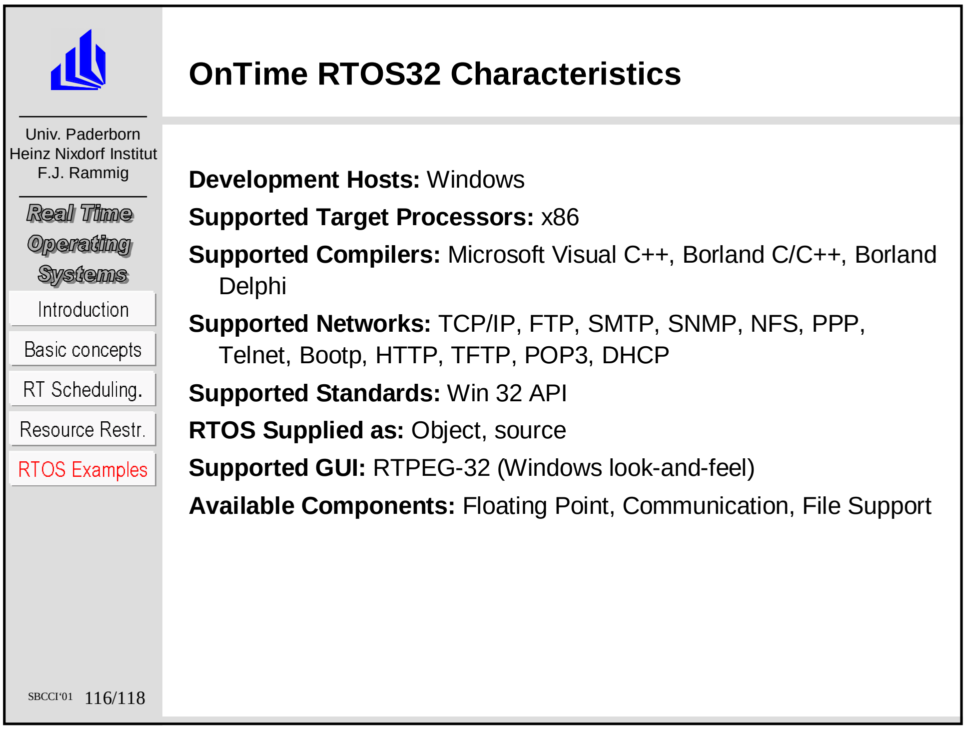
Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

OnTime RTOS32 Description

- Implements a Windows NT subset in a 16K of memory
- It fully integrates with Microsoft Visual Studio
- The CPU's memory protection features are used to guarantee that programs cannot overwrite protected data, code, or critical system tables



Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time
Operating
Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- RTOS Examples

OnTime RTOS32 Characteristics

Development Hosts: Windows

Supported Target Processors: x86

Supported Compilers: Microsoft Visual C++, Borland C/C++, Borland Delphi

Supported Networks: TCP/IP, FTP, SMTP, SNMP, NFS, PPP, Telnet, Bootp, HTTP, TFTP, POP3, DHCP

Supported Standards: Win 32 API

RTOS Supplied as: Object, source

Supported GUI: RTPEG-32 (Windows look-and-feel)

Available Components: Floating Point, Communication, File Support

Overview

Basis	LINUX				QNX		UNIX	Win	Other	
Name	**RT-Linux**	**RTAI**	**RED Linux**	**Kurt**	**QNX Neutrino**	**QNX4**	**Lynx OS**	**RTOS 32**	**Vx Works**	**RTEMS**
Multiprocess	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes*	Yes	Yes
Multiprocessor	Yes	Yes	No	No	Yes	No	Yes	No		Yes
Max internal latency	<20us	~25us	<50us	~20us			14us	5us		
Clock resolution	<1us	<1us	~5us	<1us	depends	depends	20us	100us		
Main scheduling	EDF	Priority based	On choice	Priority based	Prioritized Fifo	Prioritized Fifo	Pr. Fifo	Pr. Fifo	Priority based	Priority based
Priority inversion avoidance	No	Yes			Yes	Yes	Yes	Yes		Yes
Source/Objec	Source	Source	Source	Source	Object	Object	Source	Source	Objecte	Source

* Non preemptive schedule

Priority Based: e.g. Rate monotonic, EDF,...

URLs

Univ. Paderborn
Heinz Nixdorf Institut
F.J. Rammig

Real Time Operating Systems

- Introduction
- Basic concepts
- RT Scheduling.
- Resource Restr.
- Priority Inversion
- RTOS Examples

RT-Linux	http://www.rtlinux.org
RTAI	http://www.rtai.org
RED-Linux	http://linux.ece.uci.edu/RED-Linux/
KURT	http://www.ittc.ukans.edu/kurt/
LynxOS	http://www.lynuxworks.com/
QNX4	http://www.qnx.com
Neutrino	http://www.qnx.com
RTEMS	http://www.rtems.com
On Time RTOS32	http://www.on-time.com
VxWorks	http://www.windriver.com